University of Southern Denmark

DM852

Heuristics & Approximations Algorithms

Submitted To:
Marco Chiarandini
Lene Monrad Favrholdt
IMADA
Mathematics & Computer Science Department

Submitted By :
Alexander Lerche Falk
Narongrit Unwerawattana
Spring - Master of Computer Science

Contents

1 Introduction **2**

2 Simulated Annealing **3**
2.1 Complexity and Analysis 4

3 Ant Colony Optimization **9**

4
BOXPLOT FIGURES **10**

Appendices **11**

1 Introduction

This project shows metaheuristic algorithms, resolving the Capacitated Vehicle Routing Problem (CVRP). The difference between metaheuristics and heuristics is in the solution part. For heuristics, you are trying your best to find a solution, even though it is not optimal. The algorithm is adapted to the problem in such a greedy approach, it can get stuck in a local optimum. This is fine since it is the idea of heuristics: "just solve the problem".
Metaheuristics are less greedy and tends to be more problem independent. They accept temporary solutions and allow "bad" steps as an attempt to get a better solution. You have a local- and global optimum to keep track of the best solution found. You can say metaheuristics are exploting heuristics to avoid getting trapped.
In this metaheuristics implementation project, we have chosen to implement two algorithms for CVRP: Simulated Annealing (SA) and Ant Colony Optimization (ACO). The SA algorithm is the one we have uploaded for electronic submisison at http://valkyria.imada.sdu.dk/D0App/. The ACO algorithm is implemented but does not perform well. We will compare the algorithms with our previous heuristic / local search project and lastly, show our results of computation for our two algorithms.

— Alexander & Narongrit

2 Simulated Annealing

The Simulated Annealing (SA) algorithm is inspired from the annealing process in metal, where you are altering the physical state of the metal by heating and cooling it. The inspiration can be used in computer science as well. We can use the algorithm on CVRP by starting off by generating a solution to the problem and not "care" about the initial solutions. The better steps we are taking, and better solutions we are finding, the more careful we are going to be in finding a solution. In the beginning we allow random and bad steps to be performed, while later, we make better calculations.
We have developed an algorithm to solve CVRP using Simulated Annealing technique by relating CVRP with existed problems i.e. Bin packing problem and Knapsack problem. Using this approach, we can guarantee maximum amount of vehicles necessary required with approximation analysis on bin packing hueristic algorithm. In this demonstration, we use well studied algorithm First-Fit-Decreasing which is guaranteed to use no more than $\frac{11}{9}\mathrm{OPT} + 1$ number of required vehicles[1]. With combination of simple Traveling Salesman Problem's local search algorithm, 2-opt, we are able create the initial feasible solution as described in Algorithm 3. Next step of Simulated Annealing involved creating new feasible solution by altering current solution. Which in CVRP case, is to perform points exchange action between routes. In order to allow as much possibility of exchanges as it could without breaking capacity constrain, we choose simple algorithm to find exchanges possibility based on Knapsack problem 4. Our implementation of acceptance criterion is based on Metropolis condition which accept new solution by using probability $p = \exp[-\frac{new\_cost - best\_cost}{temp}]$, this leads to $p = 1$ when $new\_cost < best\_cost$ otherwise $p = [0, 1)$. We use geometric cooling scheme in order to update temperature every iteration the algorithm accepts new solution by multiply temperature variables with $alpha$ where $alpha = (0, 1)$. This makes p have lower probability to accept worsen solution after later iteration. All algorithms is described as following.

Algorithm 1 Simulated annealing with First-Fit-Decreasing and Knapsack combination

Require: *customers*: List of customer coordinate with it's capacity
Require: *max_cap*: Maximum capacity for each route
Require: *depot*: Depot point
Require: *is_accept*(*best_cost*, *cost*, *temp*): an acceptance criterion function returns boolean based on Metopolis condition

```
function SIMULATED_ANNEALING_FFD(temp, alpha, no_improve_limits)
    routes ← CVRP_FIRST_FIT_DECREASING(customers, depot)
    best_cost ← total cost of routes
    best_routes ← routes
    current_cost ← best_cost
    current_routes ← routes
    curr_no_improve_trying ← 0
    while True do
        if curr_no_improve_trying == no_improve_limits then break
        new_routes ← NEIGHBORS_SELECTION(current_routes)
        new_cost ← cost of new_routes
        if is_accept(best_cost, new_cost, temp) then
            temp ← temp * alpha
            curr_no_improve_trying ← 0
            update current_routes with new routes
            if new_cost < best_cost then
                update best_cost and best_routes with new routes
        else
            curr_no_improve_trying ← curr_no_improve_trying + 1
    return best_routes
```

2.1 Complexity and Analysis

By relating CVRP to other existed problem, we are able to come up with a simple algorithm and use existed technique, Simulated Annealing, to improve the solution. We first analyzing our algorithm to construct our initial routes. Using First-Fit-Decreasing to assign group customer points for a vehicle only have runtime complexity of $O(nlogn)$ but in order to decide customer point sequences for each route, we rely on local search algorithm for traveling salesman problem which dominates runtime complexity of our bin packing algorithm. Using simple 2-opt local search algorithm arrange customer point sequences for each vehicle route yields runtime complexity of $O(n^2)$.

Another key task of Simulated Annealing is the neighbor selection procedure. Our attempt is to find largest possibility of exchanging customer point between routes.

Algorithm 2 First-Fit-Decreasing for CVRP

Require: *customers*: List of customer coordinate with it's capacity
Require: *tsp_local_search*(*route*): performs traveling salesman local search and returns improved route
Require: *max_cap*: Maximum capacity for each route
Require: *depot*: Depot point
Require: *route_cost*(*route*): returns capacity of route
Ensure: *routes*: solution routes

```
function CVRP_FIRST_FIT_DECREASING(customers)
    sorted_capacity_points ← sort customer points by it's capacity
    routes ← []
    add new array with depot to routes to create new route
    for all point in sorted_capacity_points do
        is_fit ← False
        for all route in routes do
            if route_cost(route) + point.capacity ≤ max_cap then
                add point to route
                is_fit ← True
                break
        if !is_fit then
            add new route with depot and point to routes
    for all route in routes do
        route ← tsp_local_search(route)
    return routes
```

The problem is risen when given one customer point chosen by uniform distribution probability, how can we decide other candidate to make exchange action and such candidate should not leads to infeasible solution. Introducing a simple algorithm to find possibility of packing customer point into one set based on brute force algorithm for Knapsack problem, which explore for every customer points either to include in neighbor exchange candidate or not, allows us to find a list of feasible candidates to make exchange between routes. But tradeoff of this algorithm is the runtime complexity, which is $O(2^n)$ where n is number of customer points in candidate route. In order to keep runtime of each Simulated Annealing iteration small, we introduce a user-defined variable *max_points_amt* which an algorithm will use to choose number of customer points in candidate route and proceed the algorithm. This indeed reduces the number of possibility on selecting exchange neighbors candidates but in an instance where it consist of many small capacity point, without limiting such customer point might leads to very long runtime for each neighbor selection process. We conclude that for each iteration of our Simulated Annealing the runtime complexity is in $O(2^n)$ where n is number of maximum points allowed

Algorithm 3 Neighbours selection for SA

Require: *routes*: list of routes in solution contains customer coordinate and it's capacity required
Require: *max_cap*: Maximum capacity for each route
Require: *route_cap*(*route*): returns capacity of *route*
Require: *tsp_local_search*(*route*): performs traveling salesman local search and returns improved route
Ensure: *new_routes*: new solution routes

function NEIGHBOR_SELECTION(*routes*)
  repeat
    $origin\_route\_id, target\_route\_id \leftarrow$ randomly pick 2 distinct index of *routes*
    $origin\_route\_point \leftarrow$ randomly pick element from $route[origin\_route\_id]$
    $upper\_bound \leftarrow max\_cap - (route\_cap(routes[origin\_route\_id] - origin\_route\_point.capacity))$
    $lower\_bound \leftarrow route\_cap(routes[target\_route\_id]) - (max\_cap - origin\_route\_point.capacity)$
    $target\_neighbor\_set \leftarrow$ KNAPSACK_COMBINATION($route[target\_route\_id], lower\_bound, upper\_bound$)
  until $len(target\_neighbor\_set) > 0$
  $target\_neighbor \leftarrow$ randomly pick an element from $target\_neighbor\_set$
  $new\_route\_origin, new\_route\_target \leftarrow$ exchange neighbors between two route and assign to new variables
  $new\_route\_origin \leftarrow tsp\_local\_search(new\_route\_origin)$
  $new\_route\_target \leftarrow tsp\_local\_search(new\_route\_target)$
  $new\_routes \leftarrow routes$ where $route[origin\_route\_id]$ and $route[target\_route\_id]$ replaced by new routes
  return *new_routes*

to be exchange.

Algorithm 4 Knapsack Combination

Require: *customers*: List of customer coordinate with it's capacity
Require: *max_cap*: Maximum capacity for each route
Require: *depot*: Depot point
Require: *max_points_amt*: Number of maximum combination points
Require: *route_cap*(*route*): returns capacity of *route*
Ensure: *combination_set*: set of candidates combination of points in *route*

```
function KNAPSACK_COMBINATION(route, lower_bound, upper_bound)
    points ← randomly pick max_points_amt points from route but depot
    combination_set ← []
    function KNAPSACK_COMBINATION_RECUR(current_point_id, current_capacity, knapsack)
        if current_point_id == len(points) then
            if route_cap(knapsack) > lower_bound then
                add knapsack to combination_set
        knapsack_combination_recur(
            current_point_id + 1, current_capacity, knapsack)
        if current_cap + points[current_point_id] ≤ max_cap then
            add points[current_point_id] to knapsack
            current_cap ← points[current_point_id].capacity
            knapsack_combination_recur(
            current_point_id + 1, current_capacity, knapsack)
    knapsack_combination(0, 0, [])
    return combination_set
```

instance	**CHH**		**SA**		**instance**	**CHH**		**SA**	
	k	**cost**	**k**	cost		**k**	**cost**	**k**	cost
A-n32-k05	5	867	5	980	CMT01	6	604	5	553
A-n33-k05	5	743	5	661	CMT02	11	920	10	894
A-n33-k06	6	766	6	816	CMT03	8	985	8	987
A-n34-k05	6	889	5	778	CMT04	12	1196	12	1568
A-n36-k05	5	862	5	807	CMT05	17	1496	17	3406
A-n37-k05	5	741	5	669	CMT11	8	1111	7	1517
A-n37-k06	7	1112	6	949	CMT12	10	909	10	1180
A-n38-k05	6	822	5	730	Golden_01	9	6055	9	8796
A-n39-k05	5	883	5	827	Golden_02	10	9121	10	14227
A-n39-k06	6	887	6	999	Golden_03	9	12144	9	17920
A-n44-k06	6	1029	6	939	Golden_04	10	15644	10	24338
A-n45-k06	7	1014	6	978	Golden_05	5	7330	5	10954
A-n45-k07	7	1250	7	1161	Golden_06	7	9698	7	14509
A-n46-k07	7	997	7	1039	Golden_07	9	11655	9	17513
A-n48-k07	7	1232	7	1128	Golden_08	10	12835	10	19443
A-n53-k07	8	1142	7	1054	Golden_09	15	590	14	653
A-n54-k07	8	1258	7	1217	Golden_10	16	763	16	843
A-n55-k09	9	1158	9	1300	Golden_11	18	963	18	1090
A-n60-k09	9	1412	9	1394	Golden_12	20	1192	19	1199
A-n61-k09	11	1300	9	1111	Golden_13	28	972	26	1175
A-n62-k08	8	1379	8	1363	Golden_14	31	1194	30	1471
A-n63-k09	10	1764	9	1676	Golden_15	35	1501	33	1743
A-n63-k10	11	1504	10	1442	Golden_16	39	1880	37	2092
A-n64-k09	9	1530	9	1501	Golden_17	22	774	23	1289
A-n65-k09	10	1274	9	1260	Golden_18	28	1108	29	2350
A-n69-k09	10	1297	9	1266	Golden_19	34	1578	35	3279
A-n80-k10	10	1903	10	1920	Golden_20	39	2084	40	4309

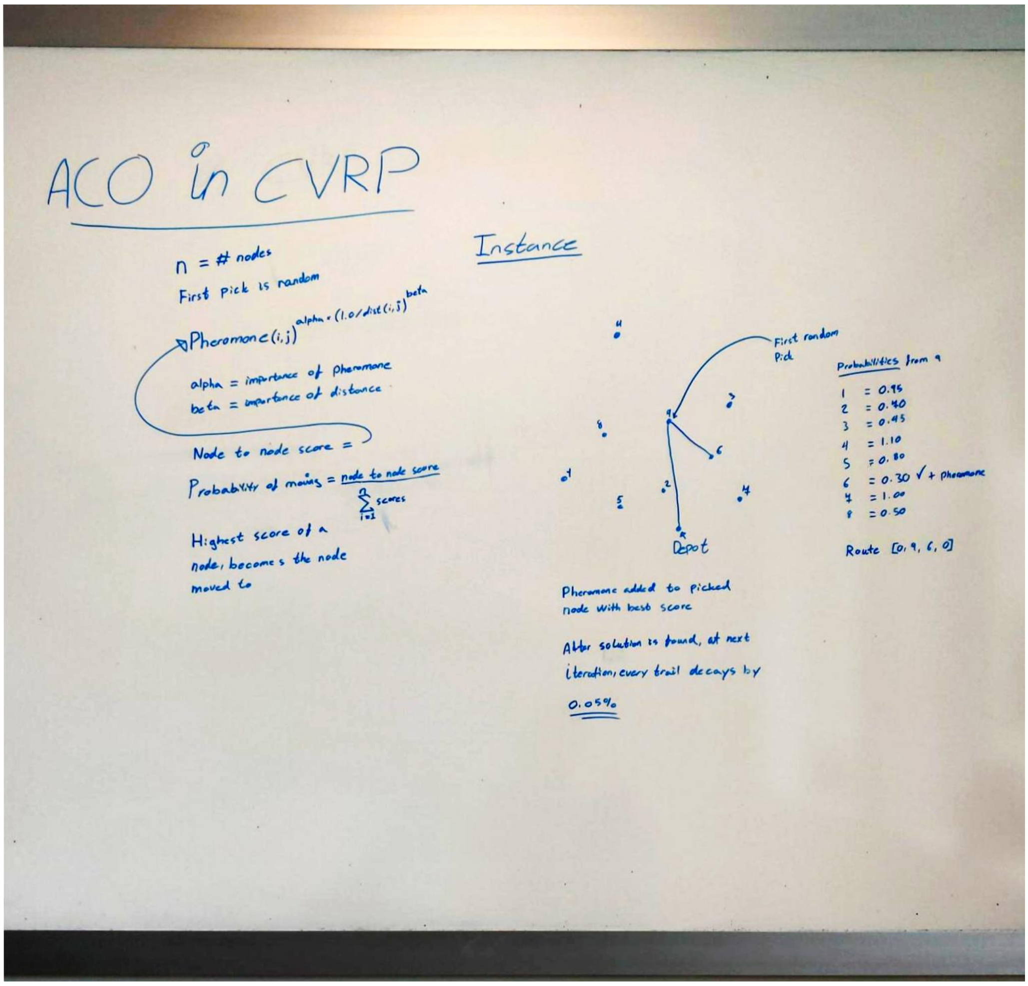
Table 1: Result comparing between Heuristic algorithm and Metahuristic(SA with $max\_points\_amt = 3, temp = 1000, alpha = 0.95$) running with time limit 240 seconds on an Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz with 4 GB allocated RAM running Ubuntu 16.04.

3 Ant Colony Optimization

The Ant Colony Optimization (ACO) algorithm comes from the evolutionary algorithms, where computer science meets nature. In this algorithm, we have led us to be inspired by how ants find food in the nature. They are starting by spreading out in some area, randomly. When they seem to find trails, which could indicate to be good trails to find food, they leave pheromone behind them. The pheromone is used by other ants to determine their probability of taking a trail. If they are standing in a cross-section and they have to choose, they are taking the path with the highest pheromone. At some point in their exploration to find the best trail to obtain food, all the ants are using the same trails to get food and get back safe home.
We can apply the logic of the ants in the CVRP as well. By letting the instance the space of which the requests are "plotted" be the area to find a solution, we can start by sending one "ant" out to a point. From this point, we are going to calculate every probability of moving to the next point the one with the highest "score". We can calulate the probability of moving by the following:
First we establish the initial pheromone levels from all the points to every counter other point:

$$M = \begin{bmatrix} 0 & 0.50 & 0.20 \\ 0.125 & 0 & 0.60 \\ 1.20 & 0.355 & 0 \end{bmatrix}$$

Figure 1: Illustration of the ACO algorithm, showing how it can be used, and how it process next moves



4 BOXPLOT FIGURES

References

[1] Minyi Yue. A simple proof of the inequality ffd(l) $\leq$ 11/9 opt(l) + 1, $\forall$l for the ffd bin-packing algorithm. *Acta Mathematicae Applicatae Sinica*, 7(4):321–331, 1991.

Appendices